**Aufgabe 3: Dreiecke zählen**

|  |  |
| --- | --- |
| Datei | bwinf aufgabe 3.exe |
|  | Getestet auf Windows 8.1/10 |

Um die Aufgabe zu lösen, habe ich zunächst einige simple Datenstrukturen (structs) definiert. Dazu gehören:

struct Linie; (beschreibt eine Linie, beinhaltet die Variablen x1, x2, y1, y2 zur Definition des Anfangs- und Endpunkts der Linien)

struct Vec2f; (dient zur Speicherung einer 2-Dimensionalen Koordinate mit Float-Werten, speichert X- und Y-Wert in den Membervariablen float x und y)

struct Vec3i; (dient zur Speicherung von 3 Integer-Werten, die drei Werte sind nach ihrer Größe geordnet (mit dem kleinsten beginnend) aus Gründen, die ich später noch erläutern werde)

Nachdem das Programm gestartet wurde, wird der Nutzer als erstes nach der txt-Datei gefragt, in der die Werte für die Linien gespeichert sind. Die Methode „std::string getFileName()“ öffnet ein Windows „IFileOpenDialog“-Fenster, bei dem der Nutzer das Verzeichnis der Datei raussuchen und die Datei so einfacher lokalisiert werden kann. Die Methode gibt dann den Dateipfad als Rückgabewert zurück. Den Inhalt der Methode habe ich aus dieser Quelle: http://www.cplusplus.com/forum/windows/169960/.

Der Dateipfad wird anschließend an die Methode „void dateiAuslesenUndDatenSpeichern(…)“ als Parameter übergeben. Außerdem wird eine Referenz zu einem „std::vector<Linie>“ übergeben. In diesem sollen am Ende alle Linien gespeichert werden.

FILE\* file;

fopen\_s(&file, fileName.c\_str(), "r");

// Zunächst wird die Datei geöffnet.

int size;

fscanf\_s(file, "%i", &size);

while (true)

{

Linie line;

// Es wird für jede Zeile der Datei (außer der Ersten) eine temporäre Linie erstellt.

int res = fscanf\_s(file, "%f %f %f %f", &line.x1, &line.y1, &line.x2, &line.y2);

// Die Werte der Datei werden Zahl für Zahl als Float-Werte in die temporäre Linie

// gespeichert.

if (res == EOF)

{

break;

}

linien.push\_back(line);

// Am Ende jeder gelesenen Zeile wird die temporäre Linie in den Linien-Vector (der alle

// Linien speichert) abgelegt.

}

Um die Linien ordentlich auswerten zu können hab ich noch zwei Funktionen implementiert:

bool linie\_A\_schneidet\_Linie\_B(Linie linie1, Linie line2) { … }

// Gibt zurück, ob sich zwei Linien schneiden.

Vec2f getSchnittpunkt(Linie line1, Linie line2)

{

float x12 = line1.x1 - line1.x2;

float x34 = line2.x1 - line2.x2;

float y12 = line1.y1 - line1.y2;

float y34 = line2.y1 - line2.y2;

float c = x12 \* y34 - y12 \* x34;

float a = line1.x1 \* line1.y2 - line1.y1 \* line1.x2;

float b = line2.x1 \* line2.y2 - line2.y1 \* line2.x2;

float x = (a \* x34 - b \* x12) / c;

float y = (a \* y34 - b \* y12) / c;

return Vec2f(x, y);

// Gibt die Koordinaten des Schnittpunktes zweier Linien in Form eines „Vec2f“ zurück

}

Die Lösung der Aufgabe kann nun also beginnen. Das Lösungsprinzip sieht wie folgt aus:

Erstmal muss heraus gefunden werden, welche Dreiecke es überhaupt gibt. Ich iteriere (x) also durch alle Linien des Linien-Vectors und iteriere (y) bei jeder Linie durch alle anderen Linien und überprüfe, ob diese sich schneiden. Wenn das der Fall ist, dann iteriere (z) ich nochmal durch alle Linien, und gucke, ob sich Linie z und y schneiden. Ist das dann der Fall, überprüft das Programm ob sich Line z mit Linie x schneidet. Schneidet sich also Linie x mit y, y mit z und z wiederrum mit x, hat das Programm ein Dreieck gefunden. Anschließend wird nur noch überprüft, ob dieses Dreieck nicht schonmal gefunden wurde. Es sollten ja keine Doppelungen vorkommen.

std::vector<Vec3i> dreiecksSpeicher;

// Zunächst lege ich einen Speicher für Dreiecke fest: Für jedes gefundene Dreieck werden hier die // Indizes der Linien, aus dem es besteht, abgelegt. Da im Vec3i die Werte immer direkt nach der // Größe geordnet sind, lassen sich zwei Vec3i leichter vergleichen, da die Reihenfolge der Linien bei // einem Dreieck keine Rolle spielen. Somit wäre (Vec3i(2,5,7) == Vec3i(7,5,2)) -> true. Damit können // wir verhindern, dass Dreiecke doppelt abgespeichert werden.

for (size\_t x = 0; x < linien.size(); x++) // x-Iteration

{

for (size\_t y = 0; y < linien.size(); y++) // y-Iteration

{

if (y != x)

{

if (linie\_A\_schneidet\_Linie\_B(linien[x], linien[y]))

{

for (size\_t z = 0; z < linien.size(); z++) // z-Iteration

{

if (z != x && z != y)

{

if(linie\_A\_schneidet\_Linie\_B(linien[y], linien[z]))

{

if (linie\_A\_schneidet\_Linie\_B(linien[x], linien[z]))

{

Vec3i tempDreieck(x, y, z);

bool dreieck\_bereits\_gespeichert = false;

// Überprüfung, ob das Dreieck schon gefunden wurde

for (size\_t w = 0; w < dreiecksSpeicher.size()

&& !dreieck\_bereits\_gespeichert; w++)

{

if (dreiecksSpeicher[w] == tempDreieck)

{

dreieck\_bereits\_gespeichert = true;

}

}

if (!dreieck\_bereits\_gespeichert)

{

// Speicherung des gefunden Dreiecks

dreiecksSpeicher.push\_back(tempDreieck);

}

}

}

}

}

}

}

}

}

std::cout << std::endl << "Es sind " << dreiecksSpeicher.size() << " Dreiecke vorhanden!" << std::endl << std::endl;

Nachdem alle möglichen Dreiecke gefunden sind, geht es um die Präzise Ausgabe der Schnittpunkte. Dafür benutzt das Programm die Methode getSchnittpunkt(…) und gibt die Schnittpunkte aller Linien der Dreiecke hintereinander in der vorgegebenen Form aus. In der Aufgabe stand zwar, die Schnittpunkte sollten in einer txt-Datei gespeichert werden, aber wegen Zeitdruckes und da es eher nebensächlich ist, habe ich mich für diese Form der Ausgabe entschieden.

std::cout << "x\ty\tx\ty\tx\ty" << std::endl;

for (size\_t x = 0; x < dreiecksSpeicher.size(); x++)

{

std::cout << getSchnittpunkt(linien[dreiecksSpeicher[x].x],

linien[dreiecksSpeicher[x].y]).x << "\t";

std::cout << getSchnittpunkt(linien[dreiecksSpeicher[x].x],

linien[dreiecksSpeicher[x].y]).y << "\t";

std::cout << getSchnittpunkt(linien[dreiecksSpeicher[x].y],

linien[dreiecksSpeicher[x].z]).x << "\t";

std::cout << getSchnittpunkt(linien[dreiecksSpeicher[x].y],

linien[dreiecksSpeicher[x].z]).y << "\t";

std::cout << getSchnittpunkt(linien[dreiecksSpeicher[x].z],

linien[dreiecksSpeicher[x].x]).x << "\t";

std::cout << getSchnittpunkt(linien[dreiecksSpeicher[x].z],

linien[dreiecksSpeicher[x].x]).y << "\t";

std::cout << std::endl;

}

**Aufgabe 1: Zimmerbelegung**

|  |  |
| --- | --- |
| Datei | bwinf aufgabe 1.exe |
|  | Getestet auf Windows 8.1/10 |

Um die Aufgabe zu lösen, habe ich zunächst einige simple Datenstrukturen (structs) definiert. Dazu gehören:

struct Person; (umfasst den Namen einer Person, seinen Index im Personenspeicher, alle Namen der Person mit der er in ein Zimmer möchte und nicht möchte und alle Indices der Personen mit der die Person in ein Zimmer möchte und nicht möchte.)

struct Zimmer; (enthält einen „std::vector<unsigned int> personenIDs“, welcher die Indices aller Personen enthält, die sich in dem Zimmer befinden.)

Nachdem das Programm gestartet wurde, wird der Nutzer als erstes nach der txt-Datei gefragt, in der die Werte für die Personen gespeichert sind. Die Methode „std::string getFileName()“ öffnet ein Windows „IFileOpenDialog“-Fenster, bei dem der Nutzer das Verzeichnis der Datei raussuchen und die Datei so einfacher lokalisiert werden kann. Die Methode gibt dann den Dateipfad als Rückgabewert zurück. Den Inhalt der Methode habe ich aus dieser Quelle: http://www.cplusplus.com/forum/windows/169960/.

Der Dateipfad wird anschließend an die Methode „void dateiAuslesenUndDatenSpeichern(…)“ als Parameter übergeben. Außerdem wird eine Referenz zu einem „std::vector<Person> personen“ übergeben. In diesem sollen am Ende alle Personen gespeichert werden.

void dateiLesenUndDatenSpeichern(std::string fileName, std::vector<Person>& personen)

{

std::ifstream ifStream (fileName);

std::string line;

Person tempPerson;

// Instanzierung einer temporären Person

bool erstePersonFertig = false;

while (std::getline(ifStream, line))

{

std::vector<std::string> tempVec;

split(line, tempVec, ' ');

if (tempVec.size() != 0)

{

if (tempVec[0] == "+")

{

for (size\_t i = 1; i < tempVec.size(); i++)

{

tempPerson.proPersonen.push\_back(tempVec[i]);

// Alle von der momentanen Person im Zimmer gewollten // Personen werden abgespeichert

}

}

else if (tempVec[0] == "-")

{

for (size\_t i = 1; i < tempVec.size(); i++)

{

tempPerson.conPersonen.push\_back(tempVec[i]);

// Alle von der momentanen Person nicht im Zimmer

// gewollten Personen werden abgespeichert

}

}

else if (line.size() > 0)

{

if (erstePersonFertig)

{

if (tempPerson.name.length() > 0)

{

personen.push\_back(tempPerson);

// Die temporäre Person wird endgültig im Personen-// Vector abgespeichert

tempVec.clear();

tempPerson.proPersonen.clear();

tempPerson.conPersonen.clear();

}

}

tempPerson.name = line;

erstePersonFertig = true;

}

}

}

if (tempPerson.name.length() > 0)

{

personen.push\_back(tempPerson);

}

}

Da ja momentan nur die Namen aller Personen als Strings gespeichert sind, und es unpraktisch werden würde später mit diesen zu arbeiten, wird das Programm alle Namen (also der Name jeder Person und die Namen der gewollten/nicht gewollten Personen) in Ids umwandeln. Eine ID ist in diesem Fall einfach der Index der Person im Personen-Vector.

Für diesen Zweck hat der struct Person eine Member-Funktion, namens InIDsUmwandeln(…). Diese nutzt die Funktion std::find\_if(…), um aus dem Personen-Vector den Index der Person mit dem gesuchten Namen zu ermitteln. Somit sucht die Funktion zunächst den Index der eigenen Person heraus und speichert diesen in „unsigned int eigeneID“, danach iteriert die Funktion durch die Vektoren „proPersonen“ und „conPersonen“ und speichert die gefunden Indices in die Vektoren „proPersonenIDs“ und „conPersonenIDs“.

Nachdem alles korrkekt gespeichert ist, kann es mit der Lösung des eigentlichen Problems losgehen:

Zunächst wird erstmal ein Vektor zum Speichern der Zimmer definiert.

std::vector<Zimmer> zimmer;

// Das Programm iteriert einmal durch alle Personen durch, um die Bedürfnisse aller Personen

// zu erfüllen

for (size\_t i = 0; i < personen.size() && aufteilungMöglich; i++)

{

if (personen[i].zimmerID == -1)

{

// Ist die Person „i“ noch in keinem Zimmer, wird ein Zimmer nur für diese erstellt

Zimmer tempZimmer;

tempZimmer.personenIDs.push\_back(i);

zimmer.push\_back(tempZimmer);

personen[i].zimmerID = zimmer.size() - 1;

}

// Danach wird durch den proPersonen-Vektor der Person „i“ iteriert

for (size\_t z = 0; z < personen[i].proPersonenIDs.size() && aufteilungMöglich; z++)

{

if (personen[personen[i].proPersonenIDs[z]].zimmerID == -1)

{

// Wenn die Person-„z“ in noch keinem Zimmer ist wird folgendes gemacht:

bool personen\_Im\_Zimmer\_Akzeptieren\_Z = true;

// Zuerst wird überprüft, ob eine der Personen im Zimmer der Person-„i“ die Person-„z“ im

// conPersonen-Vector hat (also nicht mit ihr in einem Zimmer sein will

for (size\_t w = 0; w < zimmer[personen[i].zimmerID].personenIDs.size(); w++)

{

If (DoesVecContainUInt(personen[zimmer[personen[i].zimmerID]

.personenIDs[w]].conPersonenIDs, personen[i].proPersonenIDs[z]))

{

std::cout << std::endl << "Aufteilung gescheitert, weil "

<< personen[zimmer[personen[i].zimmerID].personenIDs[w]].name

<< " nicht mit " << personen[personen[i].proPersonenIDs[z]].name

<< " in ein Zimmer will"

<< std::endl;

personen\_Im\_Zimmer\_Akzeptieren\_Z = false;

}

}

bool person\_Z\_akzeptiert\_Personen\_Im\_Zimmer = true;

// Als nächstes wird überprüft, ob die Person-„z“ eine der Personen-Ids der Personen

// aus dem Zimmer der Person-„i“ im conPersonen-Vector hat

for (size\_t w = 0;

w < personen[personen[i].proPersonenIDs[z]].conPersonenIDs.size();

w++)

{

if (DoesVecContainUInt(zimmer[personen[i].zimmerID].personenIDs,

personen[personen[i].proPersonenIDs[z]].conPersonenIDs[w]))

{

std::cout << std::endl << "Aufteilung gescheitert, weil "

<< personen[personen[i].proPersonenIDs[z]].name

<< " nicht mit "

<< personen[personen[personen[i].proPersonenIDs[z]]

.conPersonenIDs[w]].name << " in ein Zimmer will"

<< std::endl;

person\_Z\_akzeptiert\_Personen\_Im\_Zimmer = false;

}

}

// Ist eine von den Bedingungen erfüllt, so ist eine Aufteilung nicht mehr möglich

if (!personen\_Im\_Zimmer\_Akzeptieren\_Z||!person\_Z\_akzeptiert\_Personen\_Im\_Zimmer)

{

aufteilungMöglich = false;

}

if (aufteilungMöglich)

{

// Sind alle noch zufrieden, so kann die Person-„z“ in das Zimmer von Person-„i“

// eingetragen werden.

zimmer[personen[i].zimmerID].

personenIDs.push\_back(personen[i].proPersonenIDs[z]);

personen[personen[i].proPersonenIDs[z]].zimmerID = personen[i].zimmerID;

}

}

else

{

// Ist die Person-„z“ bereits in einem Zimmer, müssen folgende Sachen überprüft werden:

unsigned int personZzimmerID = personen[personen[i].proPersonenIDs[z]].zimmerID;

bool personen\_Im\_Zimmer\_I\_Akzeptieren\_Personen\_Im\_ZimmerZ = true;

// Innerhalb der beiden for-Schleifen wird überprüft, ob eine Person im Zimmer der

// Person-„i“ eine Person aus dem Zimmer der Person-„z“ im conPersonenIDs-Vector hat

for (size\_t w = 0; w < zimmer[personen[i].zimmerID].personenIDs.size(); w++)

{

for (size\_t x = 0; x < zimmer[personZzimmerID].personenIDs.size(); x++)

{

if (DoesVecContainUInt(personen[zimmer[personen[i].zimmerID].personenIDs[w]]

.conPersonenIDs, zimmer[personZzimmerID].personenIDs[x]))

{

std::cout << std::endl << "Aufteilung gescheitert, weil "

<< personen[zimmer[personen[i].zimmerID].personenIDs[w]].name

<< " nicht mit "

<< personen[zimmer[personZzimmerID].personenIDs[x]].name

<< " in ein Zimmer will" << std::endl;

personen\_Im\_Zimmer\_I\_Akzeptieren\_Personen\_Im\_ZimmerZ = false;

}

}

}

bool personen\_Im\_Zimmer\_Z\_Akzeptieren\_Personen\_Im\_ZimmerI = true;

// Innerhalb der beiden for-Schleifen wird überprüft, ob eine Person im Zimmer der

// Person-„z“ eine Person aus dem Zimmer der Person-„i“ im conPersonenIDs-Vector hat

for (size\_t w = 0; w < zimmer[personZzimmerID].personenIDs.size(); w++)

{

for (size\_t x = 0; x < zimmer[personen[i].zimmerID].personenIDs.size(); x++)

{

if (DoesVecContainUInt(personen[zimmer[personZzimmerID].personenIDs[w]]

.conPersonenIDs, zimmer[personen[i].zimmerID].personenIDs[x]))

{

std::cout << std::endl << "Aufteilung gescheitert, weil "

<< personen[zimmer[personZzimmerID].personenIDs[w]].name

<< " nicht mit "

<< personen[zimmer[personen[i].zimmerID].personenIDs[x]].name

<< " in ein Zimmer will" << std::endl;

personen\_Im\_Zimmer\_Z\_Akzeptieren\_Personen\_Im\_ZimmerI = false;

}

}

}

// Ist eine von den Bedingungen erfüllt, so ist eine Aufteilung nicht mehr möglich

if (!personen\_Im\_Zimmer\_I\_Akzeptieren\_Personen\_Im\_ZimmerZ ||

!personen\_Im\_Zimmer\_Z\_Akzeptieren\_Personen\_Im\_ZimmerI)

{

aufteilungMöglich = false;

}

// Sind alle miteinander zufrieden, werden mithilfe der Funktion „personenVerlegen(…)“

// alle Personen aus dem Zimmer der Person-„z“ in das Zimmer der Person-„i“ verlegt

// und das Zimmer der Person-„z“ wird gelöscht.

if (aufteilungMöglich)

{

personenVerlegen(personZzimmerID, personen[i].zimmerID, zimmer, personen);

}

}

}

}

Wenn am Ende aufteilungMöglich noch true ist, dann ist Aufteilung möglich gewesen und die Zimmer sind auch bereits fertig aufgeteilt. Das letzt, was das Programm noch tut, ist die Ausgabe aller Personen in allen Zimmern.

**Aufgabe 4: Auto-Scrabble**

|  |  |
| --- | --- |
| Datei | bwinf aufgabe 4.exe |
|  | Getestet auf Windows 8.1/10 |

Nachdem das Programm gestartet wurde, wird der Nutzer als erstes nach der txt-Datei gefragt, in der die Kennzeichenkürzel gespeichert sind. Die Methode „std::string getFileName()“ öffnet ein Windows „IFileOpenDialog“-Fenster, bei dem der Nutzer das Verzeichnis der Datei raussuchen und die Datei so einfacher lokalisiert werden kann. Die Methode gibt dann den Dateipfad als Rückgabewert zurück. Den Inhalt der Methode habe ich aus dieser Quelle: <http://www.cplusplus.com/forum/windows/169960/>.

Die Kennzeichenkürzel werden letztendlich in dem „std::vector<std::string> kennzeichenListe“ gespeichert.